JFP **14** (5): 547–587, September 2004. © 2004 Cambridge University Press
DOI: 10.1017/S095679680300488X Printed in the United Kingdom

Two-level types and parameterized modules

TIM SHEARD and EMIR PASALIC
OGI School of Science & Engineering, Oregon Health & Science University, 20000 N.W. Walker Road, Beaverton, OR 97006-8921, USA
(*e-mail:* {sheard,pasalic}@cse.ogi.edu)

Abstract

In this paper, we describe two techniques for the efficient, modularized implementation of a large class of algorithms. We illustrate these techniques using several examples, including efficient generic unification algorithms that use reference cells to encode substitutions, and highly modular language implementations. We chose these examples to illustrate the following important techniques that we believe many functional programmers would find useful. First, defining recursive `data` types by splitting them into two levels: a structure defining level, and a recursive knot-tying level. Second, the use of rank-2 polymorphism inside Haskell's record types to implement a kind of type-parameterized modules. Finally, we explore techniques that allow us to combine already existing recursive Haskell data-types with the highly modular style of programming proposed here.

Capsule Review

Extensibility is a pressing issue for modern day software engineering. This paper rises to the challenge by giving two design patterns that functional programmers not only ought to consider, but really should have in their toolboxes. The pattern for two-level types is delightfully simple, and although parameterized modules are a little tougher to grasp, the extra effort spent understanding them is well worth it.

1 Introduction

Program modularization and *generic programming* are two practices developed to make the tasks of constructing and maintaining large programs easier. Program modularization breaks programs into smaller, potentially reusable fragments, each of which isolates a single key program aspect. This creates a separation of concerns which allows the programmer to separately create and change each program aspect. Generic programming is the construction of a single algorithm that works over multiple data structures. This allows the programmer to write an algorithm once and to reuse it on many different data structures.

This paper demonstrates two different techniques to implement these practices. These techniques are: two-level types, and type-parameterized modules. Two-level types define recursive `data` types by splitting them into two levels: a structure defining level, and a recursive knot-tying level. They can be used to both break programs into modular pieces, and to support generic programming. A type-parameterized module is a collection of (possibly polymorphic) algorithms or functions parameterized by a set of types. This allows a single module to be reused, simply by instantiating it at different types. Type-parameterized modules support program modularity in a dimension orthogonal to two-level types.

Both these techniques should be in the bag-of-tricks of any serious functional programmer. We illustrate these techniques using Haskell and its extensions. Two-level types support generic programming in standard Haskell without using any extensions. Type-parameterized modules require the commonly available extensions of rank-2 polymorphism, and multi-parameter type classes. In this paper we use the standard extensions to Haskell available in the most popular Haskell dialects of Hugs and GHC.

1.1 Contributions

This paper is an extension a functional pearl (Sheard, 2001) that appeared in the proceedings of ICFP'01. It goes beyond that paper in several dimensions. First, it is more explicit about what constitutes a two-level type, and illustrates this technique using several simple examples that were missing from the original paper. Second, it adds another large example, namely the construction of language implementations from modular language fragments. Third, it demonstrates how two-level types can coexist harmoniously with their one-level counterparts, so this technique can be retrofitted into existing programs.

The heart of this paper consists of illustrating the techniques of type-parameterized modules, and two-level types with two large examples. These examples are both rather detailed, but that is by intention. We wanted to outline in great detail how our techniques could be used. After readers have finished the paper, we want them to feel confident that they can apply these techniques to their own domains. We wanted to leave no detail to the readers' imagination.

The first example is the modularization of a whole class of algorithms that compare two instances of the same data structure. This class contains algorithms for *unification*, *matching*, and *equality* among others. The second example is the construction of highly modular language implementations.

Roadmap. Section 2 describes the technique of two level types. Section 3 develops the first large example: a generic unification algorithm. Along the way, relevant techniques we advocate are given full exposition. Section 4 presents the second large example: modular syntax. A simple programming language implementation (interpreter, parser and pretty-printer) is developed and presented in a highly modular style. Section 5 addresses the practical issue of combining two-level type programming techniques with existing programs that were not written with these techniques in mind. Section 6 distills and discusses relevant issues presented in the paper. Finally, we conclude in section 7, and give a historical note on the development of the techniques presented in the paper in section 8.

2 Two-level types

Several advantages accrue from splitting a recursive data type into two levels. To split a data type into two levels, a single recursive data type is replaced by two related data types. For example:

```
data Tree = Tip
          | Leaf Int
          | Fork Tree Tree
```

```
data T x = Tip
         | Leaf Int
         | Fork x x

data Tree = Wrap (T Tree)

unwrap :: Tree -> T Tree
unwrap (Wrap x) = x
```

The ordinary recursive data type `Tree` on the left is replaced with the two data types `T` and `Tree` on the right. The non-recursive data type `T` is called the structure operator, since it captures the structure of the original `Tree` data type. It does this by abstracting the recursive sub-trees into the type parameter `x`. The new data type `Tree` and its single constructor function `Wrap` build `Tree`s from `T` structures whose parameterized "slots" are filled with `Tree`s. Thus `Wrap :: T Tree -> Tree`.

The advantages of this approach are that it aids both modularization and generic programming by separating the structure of the data (the `T` type) from the data type itself. This makes it possible to create and manipulate `Tree`-like things whose recursive "slots" are filled with things other than `Tree`s. For example, `(Fork True False) :: T Bool` is a `Tree`-like structure with boolean "slots". This makes it possible to write generic operators like `map :: (a -> b) ->` S `a ->` S b where S can stand for any structure operator. We will elaborate on these in the large examples that follow.

A convention that we often follow is to define *lowercase* functions with the types of the original constructor functions (from the left side definition).

```
tip :: Tree
tip = Wrap Tip

leaf :: Int -> Tree
leaf x = Wrap(Leaf x)

fork :: Tree -> Tree -> Tree
fork x y = Wrap(Fork x y)
```

This allows us to construct elements of the two-level type without explicitly using the recursive knot-tying constructor `Wrap`.

3 Unification

Unification of x and y is usually defined as finding a substitution σ such that $\sigma\,x = \sigma\,y$. The terms x and y may contain variables, and a substitution is a partial function from variables to terms (often represented as a list of pairs).

Efficient implementations of unification often rely on mutable state. We show how to *modularize* unification (and its cousins *matching* and *equality*) using Haskell. Our modularization is performed along two dimensions, abstracting over the monad in which the mutable state resides (using parameterized modules), and the structure of the terms being compared (using two-level types).

The work reported in this section has been influenced strongly by two papers. *Basic Polymorphic Type Checking* by Luca Cardelli (Cardelli, 1987), and *Using Parameterized Signatures to Express Modular Structure* by Mark Jones (Jones, 1996).

Luca Cardelli's paper describes how to implement Hindley–Milner type inference for an ML-like language. The algorithm uses destructive update to achieve an efficient implementation of unification over terms representing types. This kind of unification algorithm is extremely versatile and useful. The first author of this paper has based literally dozens of other implementations on it, unifying datatypes representing many different kinds of terms.

Mark Jones' papers describes how to implement a module system by using parameterized signatures. The idea is to define functor-like operators, as defined by ML's module system (Harper & Lillibridge, 1994), that use parameterization over types rather than sharing constraints to express sharing (Harper & Pierce, 2000). The recent addition of rank-2 polymorphism to the Hugs Haskell interpreter, and the GHC compiler, allows the encoding of modules as first class objects. Our experience with this encoding provides strong evidence that type-parameterized modules really work.

An efficient implementation of unification relies on representing variables as pointers to terms. A substitution in this case is represented by the global state of the pointers. If a variable points to null, it is said to be unbound. If the pointer is not null, then the variable is bound to the term it points to. This can be implemented in Haskell by using the state monad (`ST`) (Launchbury & Peyton-Jones, 1994). In what follows, we assume the reader has a rudimentary knowledge of Haskell, unification and the `ST` monad. These, and some additional features of Haskell, perhaps unknown to some readers, can be found in Figure 2.

Brief note on variable naming conventions. Since we use type-variables named `s` for structure operators, our choice of names for type variables in the following paragraphs may seem slightly unconventional. We use variable name `a` and so on for state threads, and variable name `x` for result types of monadic computations.

In figure 1 we give a basic transcription of Cardelli's unification algorithm into Haskell. The goal of this section is to abstract details from this implementation, and to modularize it into several orthogonal components. Grasping the details of this concrete instance of the algorithm, will make it easier to understand the abstract version we will produce. We enumerate the important parts of figure 1.

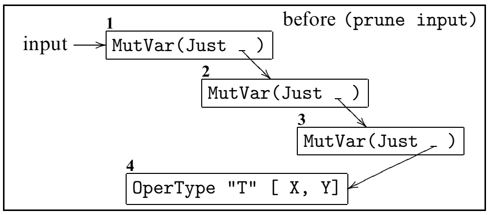
- In the `ST` monad, every function that accesses a mutable variable (to either read, write, or create a new one) has a range of type `(ST a x)`. We say that such functions have a monadic type. Each computation that access a mutable variable takes place in some thread. It is instructive to think of the type variable `a` in the type signatures of the state mutating functions as representing this thread. Thus functions whose signature contains several parameters with the same type variable `a`, must manipulate these objects in the same thread.

```
type Ptr a = STRef a (Maybe (TypeExp a))

data TypeExp a
 = MutVar (Ptr a)
 | GenVar Int
 | OperType String [ TypeExp a ]


prune :: TypeExp a -> ST a (TypeExp a)
prune t =
 case t of
   MutVar r ->
     do { m <- readSTRef r
        ; case m of
            Nothing -> return t
            Just t2 ->
              do { t' <- prune t2
                 ; writeSTRef r (Just t')
                 ; return t'}}
   other -> return t


occursInType
  :: Ptr a -> TypeExp a -> ST a Bool
occursInType r t =
  do { t' <- prune t
     ; case t' of
        MutVar r2 -> return(r==r2)
        GenVar n -> return False
        OperType nm ts ->
          do { bs <- mapM
                      (occursInType r) ts
             ; return(or bs)
             }
     }
```

```
unifyType :: TypeExp a -> TypeExp a -> ST a ()
unifyType t1 t2 =
  do { t1' <- prune t1
     ; t2' <- prune t2
     ; case (t1',t2') of
        (MutVar r1, MutVar r2) ->
          if r1==r2
             then return ()
             else writeSTRef r1 (Just t2')
        (MutVar r1, _) ->
          do { b <- occursInType r1 t2'
             ; if b then error "occurs in"
                    else writeSTRef r1 (Just t2') }
        (_,MutVar _) -> unifyType t2' t1'
        (GenVar n,GenVar m) ->
          if n==m
             then return()
             else error "different genvars"
        (OperType n1 ts1,OperType n2 ts2) ->
          if n1==n2
             then unifyArgs ts1 ts2
             else error "different constructors"
        (_,_) -> error "different types"
     }
 where unifyArgs (x:xs) (y:ys) =
         do { unifyType x y; unifyArgs xs ys }
       unifyArgs [] [] = return ()
       unifyArgs _ _ = error "different lengths"

instantiate
  :: [TypeExp a] -> TypeExp a -> TypeExp a
instantiate ts x =
  case x of
    MutVar _ -> x
    OperType nm xs -> OperType
                        nm
                        (map (instantiate ts) xs)
    GenVar n -> ts !! n
```

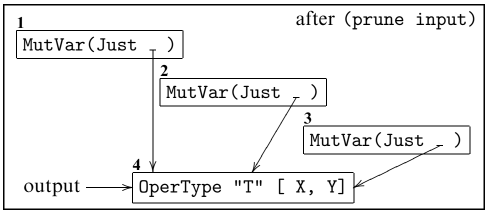
Fig. 1. Basic unification modeled after the algorithm described in Luca Cardelli's *Basic Polymorphic Type Checking*. The algorithm is transcribed from Modula-2 into Haskell.

- A TypeExp is either a variable (MutVar), a generic type (GenVar, which is used for template variables; see the last bullet of this enumeration), or a type constructor applied to a list of types (OperType). For example Int is represented by (OperType "Int" []) and Maybe x by (OperType "Maybe" [x]).
- A pointer to a (TypeExp a) is a reference cell in the ST monad that holds an object of type Maybe(TypeExp a). Nothing represents the null pointer, and Just x represents a pointer to x.
- Objects of type (TypeExp a) often contain long chains of MutVar's pointing to other MutVar's. The function prune follows such a chain, side-effecting each of the pointers in the chain to point to the element at the bottom of the chain. The value returned by prune is this last element. This is sometimes called path compression, and is illustrated below.



```
-- Monads in general
mapM :: Monad a => (b -> a c) -> [b] -> a [c]
mapM_ :: Monad a => (b -> a c) -> [b] -> a ()

-- References in the ST monad
readSTRef :: STRef a b -> ST a b
writeSTRef :: STRef a b -> b -> ST a ()
newSTRef :: a -> ST b (STRef b a)

-- Haskell's Record Syntax
-- definition
data R = R { one :: Int, two :: Bool }
-- field selection
one(R {one = 5, two = False})  --> 5
-- partial construction
R { one = 5 } --> R {one =5, two = undefined}
-- field updating
x { one = 4 }  --> R{one = 4, two = two x}
```

Fig. 2. Types of Haskell library functions dealing with monads in general, references and the ST monad, and uses of Haskell record syntax.



- The function occursInType determines if a pointer appears somewhere inside another TypeExp. Note that pointers may be changed by the call to prune. This is why occursInType has a monadic type, indicating that it might mutate state.
- The function unifyType first calls prune on its two arguments to eliminate any chains of MutVar's. The resulting objects (t1' and t2') may still be MutVar's, but if they are, then they are guaranteed to be null pointers (i.e. be references to Nothing). A simultaneous case analysis of the two resulting objects succeeds if both have the same top-level constructor, or if at least one is a variable.

  If they are both the same variable, then there is nothing to do. If two variables are matched, but are different variables, make the first variable point to the second. This is how chains of variables are created. If one is a variable, and the other is not, check if the variable occurs in the other. If it does, this is an error. If not, then make the variable point at the other object.

  If they both have the same top-level constructor then recursively analyze the substructures. This is the purpose of the local function unifyArgs.

- The function `instantiate` behaves like a substitution function, replacing every `(GenVar n)` with the `n`'th element of the substitution list `ts`. The function `instantiate` is the generic interface to templates. A template is a `GT` object containing no `MutVars`. Templates are used when one wants to unify one `GT` term with many other `GT` terms. Once a `MutVar` becomes bound, it is difficult to unbind it without jeopardizing the correctness of the unification algorithm. If it is necessary to unify a single term many times, then a template structure must be used. A template structure contains template variables (`GenVar`) in place of `MutVar` variables. Each time the template is to be used, it is instantiated by making a copy of the template, replacing each template-variable with a *fresh* type variable. The `instantiate` function provides this capability. The technique of using `Int` to represent template variables, and a list of `TypeExp` as their instantiations derives from the paper *Typing Haskell in Haskell* (Jones, 1999).

Unification using the state of updatable reference cells to encode substitutions is hard to get right. On reason is the algorithmic details of chasing and overwriting pointers. Pointer chasing is necessary to ensure the invariant that a pointer, and the thing it points to, are in some sense semantically equivalent. Every function that manipulates `TypeExps` must maintain this invariant. One way to accomplish this is to use a "copying" algorithm that removes these excess pointers as other computations are performed. Overwriting a minimal number of pointers ensures that the time behavior of the algorithm remains tractable, but adds to the complexity of programming. Because overwriting is a stateful operation, everything must be in some appropriate monad.

As the datatype representing terms becomes more complex, the control structure of the algorithm becomes more sophisticated. Terms with complex substructure require sophisticated control to make recursive calls on sub terms properly, and to combine the results of the recursive calls to build a suitable overall result.

It would be nice to separate the pointer issues from the control issues, get each one done right, and then combine them. The pointer chasing could be abstracted over different monads, and reused with multiple data structures representing different kinds of terms. The control structure could be reused when defining additional algorithms over terms other than unification. After all, *matching* (where only one of the terms can have variables), and *equality testing* have control structure remarkably similar to unification.

In the rest of this section we explain how this algorithm can be modularized in Haskell using two-level types and type-parameterized modules. We use two-level types to isolate the code dealing with structure of the term being unified into one module, and type-parameterized modules to isolate the code dealing with pointer chasing into another module. Stateful operations in a declarative language, such as Haskell, are often implemented using monads, and type-parameterized modules allow us to abstract over the particular monad being used.

3.1 Modular data

Separating the pointer algorithms from the algorithms that deal with the structure of terms, requires splitting the datatype TypeExp into several datatypes. This separates the *structure of terms* and the *use of variables* into two different datatypes. All kinds of terms which support unification will have variables, but their structure will vary according to what the terms are meant to represent. We separate terms into two levels. The first level incorporates the two kinds of variables encoded in the constructors MutVar and GenVar. The second level incorporates the role played by the constructor OperType. It abstracts the structure of all the other constructor functions of terms. The *recursive* structure of terms will be split between the two levels.

We will make this more clear by applying our technique to the same kinds of terms we used in figure 1. We split the definition of TypeExp into the two components S (for the *Structure* of terms) and GT (for *Generic Term*).

```
-- Structure operator, hence "S"
data S x = OperType String [x]

-- Generic Terms, hence "GT"
-- "s" abstracts over structure, "r" over references
data GT s r
  = S (s (GT s r))
  | MutVar (r (Maybe (GT s r)))
  | GenVar Int

type TypeExp a = GT S (STRef a)
```

We call S the structure operator because it captures the structure of the terms we are manipulating. Note how it is parameterized by x which appears in places where recursive calls to TypeExp were placed in the original definition. This is the first half of capturing the recursive structure of terms. The structure of TypeExp is quite simple, so S only has one constructor (OperType), but we shall soon see examples where the structure of terms is much richer.

The GT datatype incorporates the role of variables in terms: template variables (GenVar), and normal variables (MutVar). It abstracts over the rest of the structure of terms using the higher order parameter s. It is in the type of the S constructor function that GT captures the second half of the recursive structure of TypeExp. Note how a recursive call (GT s r) is passed to the parameterized structure operator s.

It is important to note that the parameter s to GT is a type constructor (of kind * -> *), not a type. The parameter r is also a type constructor (of kind * -> *). It abstracts over the type constructor constructing mutable references. The type constructor GT is recursively defined: both the MutVar and S constructor functions use the recursive calls to (GT s r), forwarding the recursion through the type constructors s and r.

The new version of TypeExp is an instantiation of GT, choosing for its two parameters the structure operator S, and the STRef type constructor for references from the ST monad.

Values of type `TypeExp` are constructed in two levels. An outer level consisting of one of the constructors of `GT`: `S`, `MutVar`, or `GenVar`; and in the case of the constructor function `S` an inner level consisting of the constructor function of the type constructor `S`: (`OperType`) (in general the type constructor `S` may have many constructor functions). We illustrate this in the following table:

New, two-level examples	Old, one-level examples
`GenVar 4`	`GenVar 4`
`S (OperType "Bool" [])`	`OperType "Bool" []`

`S (OperType "Maybe"`
`   [ S (OperType "Int" [])])`
	`OperType "Maybe"`
	`  [OperType "Int" []]`

This pattern of prefixing every constructor of `S` with `S` to construct a `TypeExp` is captured by employing a convention that defines a new function for every constructor of `S`. These functions have the same name as the constructor, except their initial upper case letter is made lower case.

```
operType s ts = S(OperType s ts)
```

The benefit of employing this convention is that the programmer need not always use the type constructor `S` every time a term is constructed.

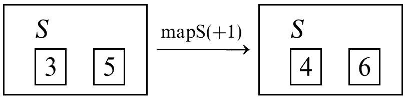
3.2 Modular abstract operations

How are functions over objects of type `(GT s r)` written if `s` and `r` are unknown? One way to do this is to assume that `s` and `r` are instances of some special classes that enumerate their general operations. For `s` we have found the following class useful.

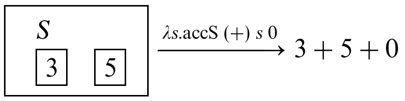
```
  class Sclass s where
    mapS   :: (x -> y) -> s x -> s y
    accS   :: (x -> y -> y) -> s x -> y -> y
    seqS   :: Monad m => s(m x) -> m(s x)
    matchS :: s x -> s x -> Maybe[(x,x)]
```

One way to understand these functions is to imagine objects of type `(s x)` as "boxes" labeled `S`, with compartments of type `x`.

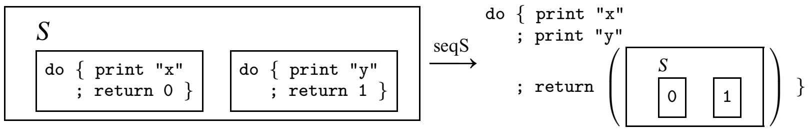
- The operation `mapS` applies a function of type `(x -> y)` to each compartment, producing a box labeled `s` with compartments of type `y`.



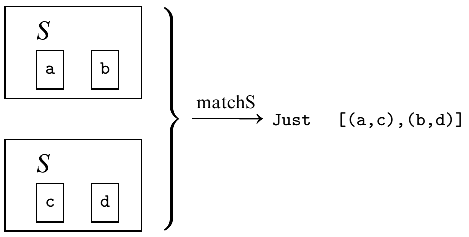
- The operation `accS` accumulates a “sum” by repeatedly applying a binary “addition” function to each of the compartments and the previous “subtotal”.



- The operation `seqS` produces a single `m`-effecting computation that produces an object of type `s x` from a box labeled `s` where the compartments are each filled by smaller `m`-effecting computations, each of which produces an object of type `x`. It does this by ordering all the sub-computations into one large computation.



- Finally, `matchS` compares the top level constructors of two `s` boxes. If the constructors match (then both boxes contain the same kind of “compartments”) it returns `Just` applied to a list of pairs, pairing corresponding compartments from each box. If the constructors do not match then it returns `Nothing`.



When `s` is the `S` datatype from our `TypeExp` example, we can use the following instance.

```
instance Sclass S where
  mapS f (OperType s xs) = OperType s (map f xs)
  accS acc (OperType s xs) ans = foldr acc ans xs
  seqS (OperType s xs) = do {xs’ <- order xs; return(OperType s xs’)}
    where order [] = return []
          order (x:xs) = do { x’ <- x
                            ; xs’ <- order xs
                            ; return (x’:xs’) }
  matchS (OperType s xs) (OperType t ys) =
    if s==t && (length xs)==(length ys)
       then Just(zip xs  ys)
       else Nothing
```

3.3 Using the abstract operations

It is now possible to write the `instantiate` function without knowing the structure of `s` at all.

```
instantiate :: Sclass s => [GT s r] -> GT s r -> GT s r
instantiate ts x =
  case x of  S y -> S(mapS (instantiate ts) y)
             MutVar _ -> x
             GenVar n -> ts !! n
```

In the first clause of the case expression, the variable `y` has type `(S (TypeExp a))`. The `mapS` function applies recursive calls of `instantiate` to each of the compartments, obtaining another object of type `(S (TypeExp a))` that is wrapped by `S` into the final answer with type `(TypeExp a)`. It is possible to write many functions over `(GT s r)` without actually knowing the exact structure of its `s` substructure. The type of `instantiate`, `(Sclass s => [GT s r] -> GT s r -> GT s r)`, makes precise the requirement that not any structure operator will do. Only those `s` that are a member of the class `Sclass`.

To write the functions `occursInType` and `unifyType` additional structure must be known about the type constructor `r`. This can be captured using two additional classes, that abstract over the operations on references.

```
class EqR r where
  sameVarR :: r x -> r x -> Bool


class Monad m => Rclass r m where
  writeVarR :: r x -> x -> m()
  readVarR  :: r x -> m x
  newVarR   ::  x -> m(r x)
```

The class `EqR` captures that references must be comparable for equality. The class `Rclass` aggregates the operations that create, read, and write references into a single class with a common multi parameter constraint. `Rclass` is multi parameter class, because it relates two type constructors, `r` the reference type constructor, and `m` the monad in which it operates.

The function `prune` over `GT s r` objects can be defined in exactly the same manner as it was over `TypeExp` objects. The definition found in Figure 1 is identical to the new definition, so we omit it here. This is possible because `prune` returns a default action on all non-`MutVar` objects, so the changes in the structure of terms is not noticed. Of course it has a different type:

```
 prune :: (Rclass r m) => GT s r -> m
(GT s r)
```

With these classes in place the function `occursInType` can be written in a generic manner.

```
occursInType :: (Rclass r m, Sclass s, EqR r) =>
          r (Maybe (GT s r)) -> GT s r -> m Bool
```

```
occursInType r t =
  do { t' <- prune t
     ; case t' of
         MutVar r2 -> return(sameVarR r r2)
         GenVar n -> return False
         S x -> do { bs <- seqS(mapS (occursInType r) x)
                   ; return(accS (||) bs False) } }
```

To determine if reference `r` occurs in term `t` prune away any leading chain of `MutVar`'s. If the result of the prune is `MutVar r2` then `r` occurs in `t` only if `r` and `r2` are the same reference. If the result is an `S` structure, `x`, then use the generic operators. First apply `occursInType` to all the compartments in `x`. The operation `mapS` performs this task. This produces an `S` structure filled with monadic computations, each of which returns a `Bool`. Applying the `seqS` operation to the `S` structure produces a larger computation returning an `S` structure filled with booleans. This structure is bound to `bs`. All the `Bool`s inside `bs`'s compartments can then be logically or-ed together (using `accS`) to produce the final result.

The function `unifyType` can be made general in a similar manner. Replace each call to reference equality with one to `sameVarR`, `readSTRef` with `readVarR`, `writeSTRef` with `writeVarR`, and `newSTRef` with `newVarR`. Finally, replace the case clause that compares two `OperType` constructors with some generic code written in terms of the operations of the `Sclass` class. The important difference are summarized here:

```
unifyType :: (Rclass r m, Sclass s, EqR r) =>
             (GT s r, GT s r) -> m ()
unifyType (t1,t2) =
  do { t1' <- prune t1
     ; t2' <- prune t2
     ; case (t1',t2') of
        . . .
         (S x, S y) ->
           case matchS x y of
             Nothing -> error "different constructors"
             Just pairs -> mapM_ unifyType pairs
        . . .
     }
```

If we are unifying two `S` structures, determine if their top level constructors are the same using `matchS`. If they don't match (`Nothing` is returned) then unification fails. If they do match, then `pairs` is a list of corresponding sub components. Recursively, unify each pair and succeed only if all are successful. The function `mapM_ :: Monad m => (b -> m c) -> [b] -> m ()` is part of the standard monad library and performs exactly the task necessary.

3.4 Why type-parameterized modules?

The solution above makes heavy use of classes. So heavy, in fact, that there are serious questions about its scalability. The type of `unifyType` is cluttered with three class constraints: `(Rclass r m, Sclass s, EqR r) => (GT s r, GT s r) -> m ()`. And, this type doesn't explicitly mention the implicit constraint `Monad m` that is implied by `Rclass r m`. When a function's type is prefixed by too many class constraints it becomes hard for the programmer to grasp the full meaning of the type.

Having many class constraints also has implications for program maintainability. As a means of documentation, it is standard practice for Haskell programmers to annotate functions with their types, even though their types could be inferred. Such annotations become hard to maintain if the number of class constraints gets large.

We have built richer implementations than the one described here. These implementations introduce recoverable errors, add a return type for `unifyType` other than `()`, and add a notion of generalization (a way of creating templates from terms with unbound variables). Each of these extensions adds an additional class or two. The types of our functions become so constrained with these additional classes that they become unreadable. The types also become so general that type inference often fails to assign a unique type to each function, requiring explicit typing annotations.

What is needed is one big "class-like" thing that abstracts over all the types `s`, `r` and `m`, and all their operations at once. Something like:

```
aBetterKindOfClass RSclass r m s where
  sameVarR  :: r x -> r x -> Bool
  writeVarR :: r x -> x -> m()
  readVarR  :: r x -> m x
  newVarR   :: x -> m(r x)
  seqS      :: s(m x) -> m(s x)
  mapS      :: (x -> y) -> s x -> s y
  accS      :: (x -> y -> y) -> s x -> y -> y
  matchS    :: s x -> s x -> Maybe[(x,x)]
```

Unfortunately, in Haskell such a class is impossible to define. The desire for type inference requires that every method in every class mention all of the abstracted variables; in this example, all of `r`, `m`, and `s`, which is just not the case. The recent suggestion (Jones, 2000) of using functional dependencies to partially alleviate this restriction is still not strong enough for the example above.

For example, a functional dependency `r -> s` or `m -> s` would allow us to infer a correct type for the method `newVar` above (which does not mention `s` in its type). Unfortunately, however, in the example above, there is *no* such dependency: the structure functor `s` is uniquely determined by neither the monad type `m`, nor the reference type `r`.

Alternatively, doing without class constraint inference, and requiring explicit type annotations at every overloaded function, could also alleviate this problem.

But class constraint inference is so useful in other contexts, that this is hardly a credible solution. What is needed is something along the lines suggested by Mark Jones in his paper *Using Parameterized Signatures to Express Modular Structure* (Jones, 1996).

Fortunately, we need not wait for a new version of Haskell with such a module system built in. We can build this functionality ourselves using an existing extension to Haskell: rank-2 polymorphism.

Without rank-2 polymorphism, universal quantification must be completely "outside" all other types. For example in the type of `(++) :: forall a . [a] -> [a] -> [a]`, the `forall` quantifies the whole type. With rank-2 polymorphism we can place the quantifier "inside" other type constructors. For example: `runST :: forall a . (forall b. ST b a) -> a`. Here the outermost quantifier `(forall a)`, quantifies the whole expression, but the `(forall b)` quantifies only the back end of the arrow type. Rank-2 polymorphism admits functions, like `runST`, that require (and make use of) polymorphic parameters. Of course the type of such functions must be explicitly declared by the programmer (Odersky & Läufer, 1996), as they cannot be inferred. These type declarations are necessary since type inference of rank-2 polymorphic types is in general undecidable.

This extension also applies to constructor functions inside of `data` declarations. They can be given polymorphic components. Using this technique we can express the structure we require.

```
data RSclass s r m = RSclass
  { sameVarRS   :: forall x. r x -> r x -> Bool
  , writeVarRS :: forall x. r x -> x -> m()
  , readVarRS   :: forall x. r x -> m x
  , newVarRS    :: forall x. x -> m(r x)
  , seqRS       :: forall x. s(m x) -> m(s x)
  , mapRS       :: forall x y. (x -> y) -> s x -> s y
  , accRS       :: forall x y. (x -> y -> y) -> s x -> y -> y
  , matchRS     :: forall x. s x -> s x -> Maybe[(x,x)]
  , errorRS     :: forall x . String -> m x
  }
```

The `RSclass` datatype definition plays the role of a type-parameterized signature. It specifies an aggregation of (possibly polymorphic) functions parameterized by a set of types. An object of type `RSclass` plays the role of a module. Every such object will specialize the type-parameters to some concrete types. A function from `RSclass` to some other type-parameterized signature plays the role of an ML-style functor.

In `RSclass` we have added an `errorRS` field to the operations we have previously discussed. In figure 1 the use of Haskell's `error` function makes aborting the program the only possible response to errors. The `errorRS` field allows the possibility that generic operations, such as `unifyType` can handle errors in some more graceful manner. Under this scheme, an instance declaration corresponds to a value of type `RSclass`, instantiated at real types for `s`, `r`, and `m`.

```
rs :: RSclass S (STRef a) (ST a)
rs = RSclass
  { sameVarRS = (==) :: STRef a b -> STRef a b -> Bool
  , writeVarRS = writeSTRef
  , readVarRS = readSTRef
  , newVarRS = newSTRef
  , seqRS = seqS
  , mapRS = mapS
  , accRS = accS
  , matchRS = matchS
  , errorRS = error
  }
```

Here the "STRef" functions are the library functions whose types were given in figure 2, and seqS, mapS, accS, and matchS are defined exactly as they were in the Sclass instance for S in Section 3.2. We have instantiated errorRS as error but other "instances" could use a different error function.

The collection of functions we wish to generate in a generic manner can also be aggregated into a data structure.

```
  data GTstruct s r m =
    B { unifyGT :: GT s r -> GT s r -> m ()
      , occursGT :: Ptr s r -> GT s r -> m Bool
      , instanGT :: [GT s r] -> GT s r -> m(GT s r)
      , pruneGT :: GT s r -> m(GT s r)
      }
```

Here the record fields of the GTstruct are populated with generic versions of the functions UnifyType, OccursInType, instantiate, and prune. The aggregate GTstruct structure can be generated by a function with type Monad m => RSclass s r m -> GTstruct s r m. This function can be written once, and applied to different RSclass objects to generate unification structures for many different term types. It can also be instantiated on different monads. We call such a function makeUnify, and a skeleton for it is given below:

```
makeUnify :: Monad m => RSclass s r m -> GTstruct s r m
makeUnify lib =
  B { freshGT = freshVar
      . . .
    }
  where freshVar = do { r <- newVarRS lib Nothing
                      ; return (MutVar r) }
         . . .
```

Note the explicit parameter lib of type RSclass to makeUnify, and the explicit application of the field selector newVarRS to lib. Using the class system, rather than encoding our own type-parameterized modules, allows this parameterization and selection to be implicit, but allows only a single instance at each type, and admits all the disadvantages enumerated in Section 3.4. In Appendix A we have included the complete code for `makeUnify`. In the appendix we have also enriched the `GTstruct` to include operations for matching, equality, and several other generic operations as well.

3.5 A richer monad

In this section we illustrate how a different monad can be used to instantiate the `RSclass` structure. A obvious choice might be to use the `IO` monad, since it too supports references. Instead, we show how to extend the monad in a different direction.

Many programs must recover from failed unification. We can accommodate this in several ways. One way, that we do not illustrate here, is to design generic unification algorithms with type: `term -> term -> M ans`, where `ans` is some type other than `()`. Thus unification can return an interesting result, that indicates what happened. Types like `Bool` , `[ Failure ]`, or `Maybe ErrorMessage` spring to mind. We have done this, and it requires adding an `answer` type to the type-parameters of `RSclass` and adding several new operations to the class, as well as making slight modifications to `makeUnify`.

A more interesting way to accommodate this is to design a richer monad that models failure, and to instantiate `makeUnify` with an instance that uses this richer monad instead of the `ST a` monad.

```
data Error = E String
newtype EM a x = EM (ST a (Either Error x))

instance Monad (EM a) where
  return x = EM(return(Right x))
  (>>=) (EM z) g =
    EM(do { c <- z
          ; case c of
              Left e -> return(Left e)
              Right x -> let EM w = g x in w
          })

runEM :: (forall b. EM b a) -> Either Error a
runEM x = let EM z = x in runST z
```

The `(EM a)` monad is a simple extension to the `(ST a)` monad. The possibility of failure is accommodated by returning `Either` an error (`Left e`) or a normal answer (`Right x`). The bind `(>>=)` operator of the monad propagates failure in its first argument without evaluating its second argument. We lift the `runST` operation to the `EM` monad with the function `runEM`. We add two new operations to the `(EM a)` monad, `raise`, and `handle`. The operation `raise` introduces an error, and `handle` catches an error, and then begins a new computation.

```
raise :: String -> EM a x
raise s = EM(return(Left (E s)))

handle :: (EM a x) -> (EM a x) -> (EM a x)
handle (EM x) (EM y) =
  EM (do { x’ <- x
         ; case x’ of
            Left _ -> y
            Right x -> return(Right x)})
```

3.5.1 Using the richer monad

The `RSclass` aggregates nine separate operations. We can define a partially defined `RSclass` object, where only the five reference and monad operations have been filled in. The laziness of Haskell makes this possible. Later in the program source we can construct many other instances of `RSclass` objects, that fill in the holes with the missing operations.

```
readVar r = EM(do { a <- readSTRef r; return(Right a) })
writeVar r x = EM(do { a <- writeSTRef r x; return(Right ()) })
newVar x = EM(do { r <- newSTRef x; return(Right r) })

emPartial :: RSclass s (STRef a) (EM a)
emPartial = RSclass
  { sameVarRS = (==)::STRef a b -> STRef a b -> Bool
  , writeVarRS = writeVar
  , readVarRS = readVar
  , newVarRS = newVar
  , errorRS = raise
  }
```

Note that `emPartial` is still polymorphic in its `s` parameter. We can use the field updating syntax of Haskell to make more complete copies of `emPartial` at many different instantiations of `s`. We do this in the definition of `commandClass` in section 3.6 below.

3.6 A richer term-structure

In this section we illustrate instantiating our general framework on a type for representing terms. We consider a type of terms representing commands in a simple imperative language. We build the `S` structure operators for this type, instantiate our general framework with the `(EM a)` monad of the previous section and illustrate the use of the framework to build a simple transformation system over commands.

```
data C x
  = If Exp x x        --  if e then s2 else s1
  | While Exp x       --  while e do s
  | Begin x x         --  { s1 ; s2 }
  | Skip              --  {}
  | Assign Var Exp    --  x := e

-- the lowercase constructor convention

ifc e x y = S(If e x y)
while e x = S(While e x)
begin x y = S(Begin x y)
skip = S Skip
assign v e = S(Assign v e)

type Command a = GT C (STRef a)
```

In Appendix B we give the definitions of the `S` structure operators `seqC`, `mapC`, `accC`, and `matchC`. These are easy to define, and in fact, could be generated automatically given the right tools (Hinze, 2000b; Hinze, 2000a). To build an `RSclass` instance using the `(EM a)` monad reference operators, we simply use the record update syntax on the partially defined `RSclass` value `emPartial` defined in the previous section.

```
commandClass :: RSclass C (STRef a) (EM a)
commandClass = emPartial { seqRS = seqC, mapRS = mapC
                         , accRS = accC, matchRS = matchC }
```

Our transformation system will implement simple rewrites over terms. For example, rewrites like `(if True then x else y)` $\longrightarrow$ `x`. To build the machinery needed, generic versions of matching, instantiation, and the creation of fresh variables will be needed. At this point we can conjure up working examples of these and several other generic operators, simply by using the generic `makeUnify` function from Appendix A.

```
B{ matchGT = match, instanGT = instan
 , freshGT = fresh, tofixGT = toFix
 , fromfixGT = fromFix } = makeUnify commandClass
```

Two things to note here. First, when using a record pattern on the left-hand-side of a binding, it is the variables to the right of the equals (=) that are being defined. In this example, those variables being bound are `match`, `instan`, `fresh`, etc. Secondly, several of the field names in this example are derived from the definition in Appendix A, rather than the one in the running text.

A transformation system is defined by rewrite rules, and an engine that applies those rules. A simple system for the command language can be built succinctly using our framework. First, define the structure of rules:

```
data Rule a = R Int (Command a) (Command a)
```

A rule is an `Int` and a pair of `(Command a)` objects. The `Int` represents the number of template variables in the rule, and the pair of `(Command a)` objects represent the left- and right-hand sides of the rewrite rule it encodes. We use this representation to simplify the implementation. The `(Command a)` objects have template variables where the rule may match any sub-command. We give some example rules below.

```
(w,x,y,z) =(GenVar 0,GenVar 1,GenVar 2,GenVar 3)

r1,r2,r3,r4 :: Rule a
r1 = R 3 (begin w (begin x y)) (begin (begin w x) y)
r2 = R 2 (ifc True w x) w
r3 = R 2 (ifc False w x) x
r4 = R 1 (while False x) skip
```

To apply a rule we instantiate the template variables with real variables, apply the matching procedure to the left-hand side of the rule and the term being transformed. If the match succeeds, then the instantiated right-hand side contains the result of the match.

```
rewrite (R n lhs rhs) x =
  handle (do { freshvs <- sequence(take n (repeat fresh))
             ; lhs' <- instan freshvs lhs
             ; match lhs' x
             ; instan freshvs rhs })
         (return x)
```

If the matching fails, then the whole `(do ... )` fails. The failure is captured by `handle`, and the original term is returned unchanged.

3.7 Escaping the monad

Monads (`M`) with mutable references have the unfortunate problem that there is no simple way to transform a computation of type `(M x)` into a value of type `x`. This restriction is imposed because it ensures that no reference escapes its scope.

It first appears, that because generic types `(GT s r)` have references inside them, that once inside the monad there is no hope of ever escaping the monad to produce "pure" values. If the monad is built on top of the `IO` monad this will always be the case, but if the monad is built on top of the `ST` monad (like the `EM` monad) this need not be so. Computations with type `(ST a x)` that are completely polymorphic in the thread type variable `a` can be converted into values of type `x` using the function `runST`.

Fortunately many terms are completely polymorphic in the state thread variable `a`. All top-level program constants with type `(GT C (STRef a))`, and larger constants derived from them, are always polymorphic in the state thread, a. Witness the type of `r1 :: Rule a`.

An important role of the `GenVar` constructor, and the `instanGT` generic function is to allow the construction of templates. Templates, because they contain no `MutVar` constructors, are always completely polymorphic. Templates can be used to create non-polymorphic instances (with `MutVar` constructors) by the use of the pattern illustrated in the function `rewrite` in Section 3.6: create a list of fresh type variables and instantiate the template using the function `instan`.

Polymorphic terms can also be constructed algorithmically, by a parser for example, if the algorithms never use the `MutVar` constructor. A term completely polymorphic in its thread variable can be extracted from the monad using `runST` in the `ST` monad or its equivalent (such as `runEM`) in other monads. In order to do this we need a form of terms that does not mention the thread variable.

If we know a generic term has no `MutVar` or `GenVar` constructors we can turn it into a type with similar structure. The `Fix` type constructor, like the `GT` type constructor, takes a type constructor as argument, but has a single constructor function, `Fix`. It plays the same role as the `S` constructor function.

```
newtype Fix s = Fix (s (Fix s))
```

Conversion between the two types can be made generic as well. In Appendix A, generic functions have been added to the `GTstruct` aggregate structure.

```
tofixGT :: GT s r -> m(Fix s)
fromfixGT :: Fix s -> GT s r
```

We can now illustrate a complete program.

```
try1 :: (forall a . GT C (STRef a)) -> Either Error (Fix C)
try1 x = runEM(do { x1 <- rewrite r1 x; toFix x1})

transform :: Fix C -> IO()
transform x =
  let x' = fromFix x
  in case try1 x' of
      Left (E s) -> print ("Fail: " ++s)
      Right y -> print(show x++" =\n"++show y++"\n")
```

4 Modular syntax

In this section we elaborate on our second large example: modular abstract syntax. We demonstrate how the two-level syntax approach can be used to define languages incrementally, from a number of modular *syntactic building blocks*. The essence of this approach is to break up a larger monolithic language into a number of smaller syntactic features, and then "mix and match" the features to build multiple "full" languages.

Haskell's class mechanism allows us to modularize interesting algorithms over "full" languages into pieces, one for each syntactic building block. This supports a

```
data Parser a       = Parser (State -> Consumed (Reply a))
data Consumed a     = Consumed a                --input is consumed
                    | Empty !a                  --no input is consumed
data Reply a        = Ok !a !State ParseError   --parsing succeeded with
@a@
                    | Error ParseError          --parsing failed
data State          = State { stateInput :: !Source
                            , statePos   :: !SourcePos }
type Source         = String

natural :: Parser Integer
(<|>) :: Parser a -> Parser a -> Parser a
symbol :: [Char] -> Parser [Char]
identifier :: Parser [Char]
try :: Parser a -> Parser a
many :: Parser a -> Parser [a]
stringLiteral :: Parser String
```

Fig. 3. Parsec parsing combinators used in the modular syntax example.

useful separation of concerns, along the lines usually only found in object-oriented approaches to program decomposition.

In this example we present three such algorithms, each with its own Haskell class definition. Each syntactic building block defines an instance of each class. The algorithms we use to illustrate this approach are parsing, evaluation, and pretty printing. Many other algorithms are also possible.

The class `Par a` classifies those types `a` which can be parsed. We will use Daan Leijen's Parsec library (Leijen & Meijer, 2001) of parsing combinators for Haskell throughout our parsing examples; the approach generalizes to any monadic parsing combinator library. A short description of the important Parsec parsing combinators used in this paper can be found in figure 3.

The `Par` class has two methods pp and ppS. This is necessary to correctly handle left recursion in the top-down parsing method implemented by Parsec. The method ppS parses *simple* phrases, such as variables, constants, parenthesized compound phrases, etc. The method pp parses compound language phrases. It is our hope that the reader has some experience with top-down parsers and can appreciate why this is necessary.

```
class Par x where
  pp :: Parser x
  ppS :: Parser x
```

The class `Eval e v m`, which uses Haskell's multi-parameter type classes, classifies triples of types. The type `e` represents the type of syntax, the type `v` represents the type of values into which this syntax will be evaluated, and the type constructor `m` represents the monad in which the evaluation will be computed. The standard Haskell class `Show` is used to represent those types that can be pretty-printed. We show the relevant class definitions:

```
class Monad m => Eval e v m where
  eval :: e -> m v

class Show a where
  show :: a -> String
```

The syntactic part of each building block represents the abstract syntax of that block's language features by using a structure operator `S e`. Where `S` is a type constructor abstracted over "slots" where components of the "full" language might appear. For each building block we need a mechanism to inject an element of type `S e` into the "full" language of type `e`. We capture this with the following class constraint between the blocks structure operator (`s`), and the type of the full language (`e`).

```
class Block s e where
  inject :: s e -> e
```

We shall examine three representative syntactic building blocks, define an instance of the above classes for each block, and show how these building blocks can be combined together.

4.1 Arithmetic building block

Suppose we wanted to define a simple language of arithmetic expressions. With the approach we outline in this section, it is important to note that we are not defining merely a language of arithmetic expressions, but rather a set of arithmetic *sub expressions* of some larger language of expressions. This larger language is not specified at the time of the design of the arithmetic syntactic building block, but is abstracted over by a type parameter.

The first step in implementing the arithmetic building block is to define a structure operator. This falls right into our strategy of using two-level types. This structure `Arith` takes as an argument the type of (possibly larger) expressions and generates arithmetic sub expressions from them.

```
data Arith e = Add e e   | Sub e e
             | Times e e | Int Int
```

Parsing. Now, suppose we wish to implement a parser for arithmetic expressions. The grammar we might wish to parse could be represented by the following BNF rule:

$$e ::= \cdots \mid e + e \mid e - e \mid e * e \mid n \mid \cdots$$

Note, however, that this rule is incomplete: we do not have all the productions of the set of terms e, since we do not know in advance all the syntactic building blocks with which the arithmetic expressions will be combined. We would expect, that given a way to parse the whole structure of terms e, we should be able to parse the arithmetic expressions (`Arith e`) built out of them[1]. We capture these constraints by the following class and instance definition:

```
instance (Block Arith e,Par e) => Par (Arith e) where ...
```

Now, we can define the appropriate instance of `Par` for arithmetic expressions. The instance declaration, given below, states that if sub expressions `e` are parsable and injectable, then, there is a way of constructing a parser for arithmetic expressions from the parser of `e`.

```
instance (Block Arith e,Par e) => Par(Arith e) where
   ppS = do { n <- natural; return(Int(fromInteger n)) }
   pp = do { e1 <- try ppS
           ; (plus e1 <|> minus e1 <|> mult e1)
           } <|> ppS
      where plus e1  = do {symbol "+"; e2 <- pp; return(Add e1 e2)}
            minus e1 = do {symbol "-"; e2 <- pp; return(Sub e1 e2)}
            mult e1  = do {symbol "*"; e2 <- pp; return(Times e1 e2)}
```

Note how `ppS` parses only *simple* arithmetic forms, i.e. literal integers. Parsing a *compound* arithmetic expressions starts with a *simple* expression followed by an arithmetic operator and then an arbitrary expression. This parsing strategy breaks the ambiguous, left recursive grammar above into a non-left recursive grammar with two precedence levels (simple terms at one level, and compound terms at the other). Note if this compound strategy fails we revert (`<|> ppS`) to a simple expression. Again we have ignored more complex precedence issues here for clarity of exposition.

Evaluation. The method (`eval :: Eval e v m => e -> m v`) is only applicable to (`Arith e`) if we can meaningfully combine vs using some semantic equivalents for the arithmetic operations: plus, times, etc. We can capture this constraint by requiring `v` to be a member of the standard Haskell class `Num`.

```
instance (Eval e v m,Num v) => Eval (Arith e) v m where
```

When defining functionality for `Arith` sub expressions, we must assume that the same functionality exists for the larger expressions. Thus, when combining syntactic building blocks, the addition of each block may force us to assume the existence of additional operations on the “full” type. Haskell’s class system is ideal for this purpose. Note how the instance of `Eval` for (`Arith e`) forces us to assume (`Num v`) as an additional constraint. The class system automatically accumulates and propagates class constraints as syntactic building blocks are combined: thus, completeness of the user’s implementation for any combination of syntactic building blocks is statically assured.

[1] One could object that the parsing example is somewhat contrived, since it does not address the issues of precedence, and some other common parsing features. The examples presented here are deliberately rather simple so as to demonstrate the idea. More realistic modular parsers are left as an exercise for the reader.

```
instance (Eval e v m,Num v) => Eval (Arith e) v m where
 eval (Int n) = return (fromInt n)
 eval (Add x y) = binop (+) x y
 eval (Sub x y) = binop (-) x y
 eval (Mul x y) = binop (*) x y

binop (#) x y = do { x' <- eval x
                   ; y' <- eval y
                   ; return (x' # y') }
```

Pretty-printing. Finally, the instance of `Show` is the easiest to define:

```
instance Show e => Show (Arith e) where
 show (Int n) = show n
 show (Plus e1 e2) = (show e1) ++ "+" ++ (show e2)
 . . .      . . .
```

4.2 Printing building block

Next, we define the building block used to add the functionality for printing expressions. The expression `(write "message" e)` prints the message to some standard output, followed by printing the value of the expression `e`.

$$e ::= \cdots \mid \text{write } msg\ e \mid \cdots$$

```
data Print x    = Write String x
```

Parsing. The parsing functionality, implemented by providing instances of `Par`, is quite straightforward.

```
instance (Par e) => Par (Print e) where
   pp = do { symbol "write"
           ; l <- stringLiteral
           ; e <- pp
           ; return (Write l e) }
```

Note that there are no *simple* parsers for print building blocks, so we give no definition for `ppS`.

Evaluation. More interesting is the definition of evaluation. As suggested by the definition of the class `Eval`, the evaluation function maps a piece of syntax into a monadic computation (in some monad `m`) of some value (in the value domain `v`). A `write` expression can be evaluated only if the monad has the ability to print. Thus, a new class `(Prints m v)` is defined. As in several of our previous examples, the class `Prints` forces us to assume additional constraints on the type representing values (`v`), and the monad (`m`) in which evaluation proceeds. The method `write` supports the printing of a textual representation of values (v) as a computation in the monad (m).

```
class (Monad m,Show v) => Prints m v where
  write :: String -> v -> m ()
```

Thus, the evaluator, now collects the explicit (Prints m v) constraint (as well as the implicit (Show v) and (Monad m) constraints) when it incorporates the printing building block. This is shown in the instance definition:

```
instance (Prints m v,Eval e v m) => Eval (Print e) v m where
  eval (Write message x)  =
    do { x' <- eval x; write message x'; return x'}
```

Pretty-printing. The pretty-printer is defined by simply instantiating the Show class, as in the previous building block:

```
instance Show e => Show (Print e) where
  show (Write s x) =
        "write " ++ (showString s "") ++ " = " ++ (show x)
```

4.3 Abstraction and application building block

The next building block we define is used for adding the functionality associated with lambda abstraction and application:

$$e ::= \cdots \mid \lambda x.e \mid e\ e \mid x \mid \cdots$$

The structure of the term is represented by the type constructor Lambda:

```
data Lambda  x = Lam String x | App x x | LVar String
```

An interesting design choice is *in which building block do variables reside?* Variables play important roles in many language features. Here we assume variables live in the Lambda block, but other choices are possible, in particular variables could reside in their own block. In that case the Lambda building block could be used only if the variable block were present.

Parsing. The parser building block is given by the code below.

```
instance (Block Lambda e,  Par e) => Par (Lambda e) where
  ppS = do { s <- identifier; return(LVar s) }
  pp  = do { symbol "\\"       -- parse "\ x -> x + 3"
           ; s <- identifier
           ; symbol "->"
           ; x <- pp
           ; return(Lam s x)
           } <|>
        do { es <- many2 ppS   -- parse "f x (y x)"
           ; return(app es)
           }
           where app :: Block Lambda e => [e] -> Lambda e
                 app [x,y] = App x y
                 app (x:y:z) = app ((inject (App x y)):z)

   many2 e = do { x <- e; y <- e; zs <- many e; return(x :y :zs) }
```

We assume that variable names in this language are simply identifiers as defined by the `Parsec` library.

Evaluation. Evaluation of the lambda building block requires additional constraints between the monad and value types.

- The class, `HasEnv m v`, defines methods for monadic computations with environments in which the values of variables can be looked up.

  ```
  class Monad m => HasEnv m v where
    inNewEnv :: String -> v -> m v -> m v
    getFromEnv :: String -> m v
  ```

- Also, the set of values into which we evaluate the lambda expressions must be able to make closures, i.e. to represent function values in the environment in which they were defined. The class `HasClosure m v` defines the following operations: `mkClosure`, which constructs a closure value by remembering its current environment; and `applyClosure`, takes a closure and an argument and forces the evaluation of the closure with the argument.

  ```
  class HasEnv m v => HasClosure m v where
    mkClosure :: String -> m v -> m v
    applyClosure :: v -> v -> m v
  ```

  With these constraints in hand, we can give the definition of eval

```
instance (HasClosure m v,Eval t v m) => Eval (Lambda t) v m where
  eval (Lam s x) = mkClosure s (eval x)
  eval (App e1 e2) = do { v1 <- eval e1
                        ; v2 <- eval e2
                        ; applyClosure v1 v2 }
  eval (LVar x) = getFromEnv x
```

Pretty-printing. Finally, we define an instance of `Show` to enable pretty-printing of this building block:

```
instance (Show e) => Show (Lambda  e) where
  show (Lam v e) = "\\" ++  v ++ "->"++(show e)
  show (LVar s) = s
  show (App x y) = show x ++ " " ++ show y
```

4.4 Combining the building blocks

Each building block uses a structure operator to encode the structure of that blocks abstract syntax. These structure operators are exactly the form that we used in decomposing types into two levels. Here we need to combine *several* of these structure operators all at once! This is accomplished by tying the recursive knot over multiple structures. For example, the datatype `Term` given below, combines arithmetic expressions, printing, and lambda expressions. All are tied together to make an abstract syntax for a "full" language.

```
data Term
  = Arith   (Arith Term)
  | Print   (Print Term)
  | Lambda  (Lambda Term)
```

Note that the single constructor function (`Wrap` in the `Tree` example) has been replaced by three constructor functions `Arith`, `Print`, and `Lambda`. By convention we name the constructors using the same name as the building block.

We now provide instances for the `Block` class.

```
instance Block Arith Term where
   inject x = Arith x
instance Block Lambda Term where
   inject x = Lambda x
instance Block Print Term where
   inject x = Print x
```

Now, for each operation (parsing, evaluation, and printing), we must tie together the individual operations for each building block. This is done by providing instances of `Par Term`, `Eval Term Value M`, and `Show Term`.

Parsing is the most complicated of the three, since parsing a "full" expression in the `Term` language relies on parsing one of the many forms parsable from the constituent language blocks. Because we have used the Parsec parsing combinators, the order in which we make these choices matters:

```
instance Par Term where
  -- order of parsers surrounding <|> actually matters
  ppS = (parens pp) <|> (fmap Arith ppS) <|> (fmap Lambda ppS)
  pp =  (fmap Arith pp) <|> (fmap Lambda pp) <|>
        (fmap Print pp) <|> ppS -- ppS comes last
```

The `Show` instance is quite straightforward. The "miracle" of Haskell classes, chooses the correct `Show` instance for each type of building block.

```
instance Show Term where
  show (Arith x)  = show x
  show (Print x)  = show x
  show (Lambda x) = show x
```

One might hope that the evaluation operation is as simple. It is almost so. Unfortunately, evaluation places constraints between the monad and value types. Thus we need a constrained instance.

```
instance (Monad m,Num v,HasEnv m v,HasClosure m v,Prints m v)
      => Eval Term v m where
  eval (Arith x)  = eval x
  eval (Lambda x) = eval x
  eval (Print x)  = eval x
```

Thus the hard work that remains is defining the appropriate evaluation instances for concrete types `m` and `v`. So, we declare the type of values, `Value`, and the type of monads `(M v a)`. In our concrete example, the monad in which we interpret `Terms` is be `(M Value)`: it combines the effects of environment passing and output.

```
data Value = I Integer | Fun (Value -> M Value Value)
type Maps x = [(String,x)]
data M value x = M (Maps value -> (x,String))
```

Instances of `Prints`, `HasEnv`, `HasClosure`, and `Num` for the value type `Value` and the monad type `(M Value)` must be provided to construct evaluators for the combined term. The relevant definitions are given below:

```
instance Prints Term Value where
  write message v = M h
     where h  env = (v,message++(show v))

instance HasEnv (M Value) Value where
  inNewEnv name v (M f) = M(\env -> f ((name,v):env))
  getFromEnv name = M h
     where h  env = (get name env,[])


instance HasClosure M Value where
   mkClosure var comp = M(\env ->
                          ((Fun(\v ->
                           (M (\_ -> unM comp ((var,v):env))))), ""))
   applyClosure (Fun f) v = f v

instance Eq Value where
   (I x) == (I y) = x == y
   _ == _ = error "no equality for functions"

instance Show Value where
   show (I a) = show a
   show (Fun _) = "<fun>"
```

Classes

```
class Par x where
 pp  :: Parser x
 ppS :: Parser x
class Monad m => Eval e v m
 eval :: e -> m v

class Show a where
  show :: a -> String

class Block s e where
 inject :: s e -> e

instance (Eval e v m, Num v)
      => Eval (Arith e) v m
  . . .
  . . .
  . . .
```

First-class modules

```
data ParBlock =
 ParBlock { pp  :: Parser x
          , ppS :: Parser x }
data Monad m => EvalBlock e v m =
 Eval { eval :: e -> m v }

data ShowBlock a =
  Show { show :: a -> String}

data Block s e =
 Block { inject :: s e -> e }

mkEval :: Num v =>
        (EvalBlock e v m) ->
        (EvalBlock (Arith e) v m)
```

Fig. 4. Classes vs. first-class modules for building blocks.

```
instance Num  Value where
   (I x) + (I y)  = I (x+y)
   _     + _ = error""
```

To demonstrate the parser and evaluator in action, we give a sample Hugs session:

```
Lang1> unM (eval (tp "(\\x -> \\y -> x+y) 1 2") :: M Value Value ) []
(3,"") :: (Value,String)
```

Finally, a note on the use of the class system in this example. In the unification example, we have advocated using type-parameterized modules to remove the need to use many classes. In the modular language construction example presented above, however, we have relied heavily on the use of classes. The important distinction is that, while the class mechanism merely gets in the way of a clean and modular algorithm specification for unification, here the class mechanism is actually useful in keeping track of the dependencies between language fragments. Thus, the type of a language-building block assembly can tell the programmer at a glance, which semantics features of the language implemented are necessary to execute assembled fragments.

Furthermore, we note in passing that the modular syntax example above could be developed using type-parameterized modules as well. Figure 4, summarizes the correspondence between the implementation with classes and an alternative first-class module implementation. Each class implementing a particular functionality (such as `Eval`) can be replaced by a type-parameterized module (data type). Instance declaration dependencies can be replaced by functions that transform type-parameterized modules (the last line in figure 4).

5 Retrofitting Two-Level Types

The use of two-level types for generic programming can be retrofitted into existing Haskell programs. Given a normal recursive data type like `Tree` in an existing program we can define a structure operator for that type. For example, given the recursive type `Tree`:

```
data Tree = Tip | Leaf Int | Fork Tree Tree
```

We define its structure operator `T`:

```
data T x = Tip' | Leaf' Int | Fork' x x
```

Here we use the convention that we *prime* (') the original constructor function names to obtain the constructor functions for the structure operator. To move back and forth between the structure operator type `T` and the recursive type `Tree` we define a pair of functions:

```
unwrap :: Tree -> T Tree
unwrap Tip = Tip'
unwrap (Leaf n) = Leaf' n
unwrap (Fork x y) = Fork' x y

wrap :: T Tree -> Tree
wrap Tip' = Tip
wrap (Leaf' n) = Leaf n
wrap (Fork' x y) = Fork x y
```

These functions play the role of the normal two-level knot-tying constructor functions such as `Wrap`, `GT`, `Arith`, etc. Of course, like any structure operator, `T` can be made an instance of the class `Sclass`.

```
instance Sclass T where
  mapS f Tip' = Tip'
  mapS f (Leaf' n) = Leaf' n
  mapS f (Fork' x y) = Fork' (f x) (f y)

  accS acc Tip' b = b
  accS acc (Leaf' n) b = b
  accS acc (Fork' x y) b = acc x (acc y b)

  seqS Tip' = return Tip'
  seqS (Leaf' n) = return(Leaf' n)
  seqS (Fork' x y) = do { a<-x; b<-y; return(Fork' a b)}
```

These capture patterns of recursion that can then be reused in many different functions. For example, we give two such examples which add all the `Leaf` values, and which transform a `Tree` by adding `3` to each `Leaf` value.

```
addAll (Leaf n) = n
addAll x = accS (+) (mapS addAll (unwrap x)) 0

plus3 (Leaf n) = (Leaf (n+3))
plus3 x = wrap (mapS plus3 (unwrap x))
```

Note how only one constructor is addressed in a specific manner in the functions above, and how the others are treated in a generic fashion using the methods of the class Sclass. When the original datatype has many constructors (such as an abstract syntax for a language), and individual functions treat most them in a uniform manner (such as might happen in alpha-renaming), then this can make the resulting program considerably more concise.

It is possible to abstract the wrapping and unwrapping into its own class.

```
class Gen1 flat rec | rec -> flat where
  wrap :: flat rec -> rec
  unwrap :: rec -> flat rec
```

By using Haskell's functional dependency extension (Jones, 2000) we note that there should be a unique structure operator (flat) for each recursive datatype (rec).

Now generic functions can be written for any normal recursive datatype which has a structure operator. First supply the Gen1 instances:

```
instance Gen1 T Tree where
  unwrap Tip = Tip'
  unwrap (Leaf n) = Leaf' n
  unwrap (Fork x y) = Fork' x y

  wrap Tip' = Tip
  wrap (Leaf' n) = Leaf n
  wrap (Fork' x y) = Fork x y
```

Then by using the methods of the Sclass, we can define very generic functions. For example a generic summing operator can be defined by:

```
genSum :: (Sclass a, Num b, Gen1 a c) => c -> b
genSum x = accS (+) (mapS genSum (unwrap x)) 0
```

And this can be specialized to numerous different recursive datatypes if they have been made instances of the Gen1 class.

```
addAll (Leaf x) = x
addAll y = genSum y
```

6 Discussion

We have shown how two-level types can be used to support both generic programming and modularity.

In the unification example, we demonstrated how two-level algebraic data-types, along with generic operators such as `map`, `seq`, `acc`, and `match` allow the construction of generic algorithms in Haskell without the use of any language extensions. We also demonstrated how to encode user-defined parameterized modules with the use of rank-2 polymorphism and Haskell records. Such modules allow a level of abstraction not possible using Haskell's class system since they allow arbitrary "overlapping" instances. Such flexibility comes at the cost of a few explicit type annotations.

In the Modular Syntax example we demonstrated how two-level types could be used in Haskell to achieve a modular decomposition of both data representation and functionality. This is the kind of decomposition is normally only available in an object oriented language. The use of Haskell's class system was critical in this example. First, the class system supports the modular decomposition of functionality, partitioning each operation into separate instances for each building block. It is possible for each building block, along with specific functionality to reside in a separate file. In addition, the class system was instrumental in ensuring that the blocks are correctly composed. Constrained classes provide just the right tool to track and manage the additional functionality required by each language building block.

Our exploration of these new techniques has not been without its setbacks. Using advanced features often pushes the limits of a paradigm or a language implementation. Three issues are worth discussion.

Mutual recursion. To generalize two-level datatypes to mutually recursive structures one needs to parameterize each structure operator over all of the recursive components, both direct and indirect. We can illustrate this using a simple language for Haskell-like expressions and declarations. Below, `E` is the structure operator for expressions, and `D` is the structure operator for declarations. Both `E` and `D` have parameters `e` and `d` which are used where recursive sub-components of type expression or declaration would normally be used.

```
type N = String              -- names
data E d e
  = Var N                    -- x
  | Const Integer            -- 5
  | App e e                  -- f x
  | Abs [N] e                -- \ x1 x2 -> e
  | Let [d] e                -- let x=e1 in e2
data D d e
  = Fun N [([N],e,[d])]      -- f p1 p2 = b where ds
  | Val N e [d]              -- p = e where ds

newtype Exp = E (E Decl Exp)
newtype Decl = D (D Decl Exp)
```

Since the structure operators (`E` and `D`) have more than one parameter, the generic operators `map` and `seq` must be generalized as well. For example:

```
mapD :: (a->b) -> (c->d) -> D a c -> D b d
seqD :: Monad m => D (m a) (m b) -> m(D a b)
```

Two-level types are not restricted to lazy languages like Haskell. We have used them in ML as well. In fact we have found them to scale quite well to even very large, highly mutually-recursive datatype declarations. We use two-level types to represent all the datatypes in the MetaML[2] implementation. We found them both easy to use, and advantageous in the genericity they supplied.

The following two problems were encountered using the Hugs interpreter in our initial research. Each has simple work-arounds. Further investigation has shown that neither problem occurs when using GHC.

Pattern matching polymorphic records. Using `data` types with polymorphic components as the input to functors (like `makeUnify`) worked well, but when we tried using pattern matching to bind the results of functor application, we stretched the limits of the Hugs implementation. For example consider the functor-like function `makeSeqmap` that takes a `(RSclass s r m)` as input and produces a `(T1 s m)` object as output:

```
  data T1 s m =
    T1 {seqmapGT :: forall x y .
           (x->m y) -> (s x) -> m(s y) }

  makeSeqmap :: RSclass s r m -> T1 s m
  makeSeqmap lib = T1 { seqmapGT = seqmap }
    where seqmap f x = seqRS lib(mapRS lib f x)
```

Everything works fine until we try and use the pattern matching feature of records to produce an actual instance of a `(T1 s m)` object with a component called `seqmap`. This top-level definition is not allowed:

```
T1 { seqmapGT = seqmap } = makeSeqmap rs
```

because of restrictions on the use of rank-2 polymorphism in pattern matching in Hugs. A work around for this is not to use pattern matching, but use the field selection mechanism instead.

```
lib = makeSeqmap rs
seqmap = seqmapGT lib
```

Update syntax of polymorphic records. In section 3.6 we specialized the `emPartial` structure by updating its (undefined) fields for `seqRS`, `mapRS`, `accRS`, and `matchRS`. Unfortunately, in the Hugs interpreter, the record update syntax is not implemented for records that contain explicitly-typed polymorphic components. Fortunately there

[2] See `http://www.cse.ogi.edu/PacSoft/projects/metaml/index.html`

is a work around for this as well. We can replace the elegant:

```
commandClass :: RSclass C (STRef a) (EM a)
commandClass =
  emPartial { seqRS = seqS
            , mapRS = mapS
            , accRS = accS
            , matchRS = matchS
            }
```

with the equivalent, but much less elegant:

```
addSpart (RSclass sv wv rv nv e s m a mtch) =
  (RSclass sv wv rv nv e seqC mapC accC matchC)

commandClass :: RSclass C (STRef a) (EM a)
commandClass = addSpart emPartial
```

7 Conclusion

Despite these minor complications, the techniques discussed here show great promise in writing high-level, generic, reusable programs in Haskell (with minor extensions). The two techniques demonstrated here: two-level syntax with generic operators, and type-parameterized `data` types with rank-2 polymorphic components, are both useful and complementary ideas that every functional programmer should be aware of.

The second idea could be made easier to use by incorporating it into a high-level module system based upon parameterized signatures as suggested by Mark Jones. Such a system should include the facility to define parameterized modules. They would serve the same purpose as the `makeUnify` function of this paper. A good design for parameterized modules would allow the names of components in the input module to be used directly, rather than relying on the selector function mechanism as was done in the definition of `makeUnify` in Appendix A. A well designed module system would also alleviate the last two problems discussed in the previous section.

Finally, although we have not used them in our implementation, we conjecture that work on generic (and polytypic) programming (Jansson & Jeuring, 1997; Jeuring & Jansson, 1996; Hinze, 2000a) could be combined beneficially with techniques presented in this paper. Combinators such as `seq` and `map` (and similar "boilerplate" (Lämmel & Peyton Jones, 2003)), which must be written anew for each new structure functor, could be automatically derived in a generic or polytypic setting, as functions over the structure of types.

These techniques would be particularly useful at retrofitting one-level types into a two-level type setting, since both structure functors, wrapping and unwrapping operations, as well as generic traversal operators on derived structure functors could be defined automatically.

A number of proposed systems add such programming techniques to Haskell (Winstanley, 1997; Hinze, 1999; Lämmel & Visser, 2000; Lämmel, 2002; Sheard

& Peyton Jones, 2002), and they could all be used in concert with the techniques advocated in this paper. We leave a fuller exploration of these issues for future work.

8 History

The first author was introduced to the use of two-level types by Erik Meijer in the fall of 1996, in a talk in which he used them to define catamorphisms, and other uniform control structures in Haskell.

```
data Fix f = Fix (f (Fix f))

fmap :: Functor f => (a->b) -> f a -> f b

cata :: Functor f => (f a -> a) -> Fix f -> a
cata phi (Fix x) = phi (fmap (cata phi) x)
```

This started a long period of experimentation with these ideas as a mechanism to write programs that did not depend upon the structure of the datatype they operated on. Ultimately, the use of uniform control structures, like `cata`, turned out to be too inflexible, but the generic operators, like `fmap` above, and `acc`, `match`, and `seq` turned out to be just the right stuff.

Most of these generic operators originate in the work of Johan Jeuring and his colleagues on *polytypic programming* (Jansson & Jeuring, 1997; Jeuring & Jansson, 1996). The one exception is `seq :: Monad m => s(m a) -> m(s a)`, which the first author likes to believe he independently discovered (though it's probably been around for years). Generic unification is one of the prime examples (Jansson & Jeuring, 1998) of polytypic programming, but the efficient algorithm employing mutable references has not been described.

The idea of type-parameterized modules originated from a discussion with Mark Jones about examples illustrating multi parameter type classes. Mark suggested abstracting the operations on stateful references away from the actual reference type constructor and monad type constructor.

```
class Monad m => Mutable r m where
  read  :: r x -> m x
  write :: r x -> x -> m ()
  new   :: x -> m(r x)
```

Frustration trying to retrofit some existing examples into this framework led to the techniques presented.

A number of works have attempted to build language implementation from reusable building blocks. The first publication of such an approach that the authors are aware off was made by Guy Steele (Steele, Jr., 1994). Sheng Liang has also used Haskell's class system together with monad transformers to structure semantics of modular programming languages (Liang *et al.*, 1995a; Liang *et al.*, 1995b; Liang & Hudak, 1996); a similar approach combining modular monadic syntax with staging in MetaML (Taha & Sheard, 1997) was also described by the authors (Sheard *et al.*, 1999).

Acknowledgements

The work described here was supported by NSF Grant CDA-9703218, the M.J. Murdock Charitable Trust and the Department of Defense. The authors would also like to thank Andy Gill, Andy Moran, Mark Jones, and Simon Peyton Jones for discussions about these ideas, Frank Taylor for LaTeXwizardry, and the entire *Advanced Functional Programming* class, at the Oregon Graduate Institute, in the winter of 2001, who suffered through endless versions of the unification implementation.

A makeUnify

```
data GTstruct s r m =
  B { unifyGT :: GT s r -> GT s r -> m ()
    , matchGT :: GT s r -> GT s r -> m()
    , equalGT :: GT s r -> GT s r -> Bool
    , freshGT :: m (GT s r)
    , occursGT :: Ptr s r -> GT s r -> m Bool
    , colGT :: GT s r -> m(GT s r)
    , pruneGT :: GT s r -> m(GT s r)
    , instanGT :: [GT s r] -> GT s r -> m(GT s r)
    , tofixGT :: GT s r -> m(Fix s)
    , fromfixGT :: Fix s -> GT s r
    }
makeUnify :: Monad m => RSclass s r m -> GTstruct s r m
makeUnify lib =
  B { unifyGT = unify
    , matchGT = match
    , equalGT = equal
    , freshGT = freshVar
    , occursGT = occursIn
    , colGT = col
    , pruneGT = prune
    , instanGT = inst
    , tofixGT = toFix
    , fromfixGT = fromFix
    }
  where
    -- first some common patterns
    seqmap f x = seqRS lib(mapRS lib f x)
    write r x = writeVarRS lib r x
    freshVar = do { r <- newVarRS lib Nothing
                  ; return (MutVar r) }
    prune (typ @ (MutVar ref)) =
      do { m <- readVarRS lib ref
         ; case m of
         Just t ->
           do { newt <- prune t
              ; write ref (Just newt)
              ; return newt }
         Nothing -> return typ}
prune x = return x
col x =
  do { x' <- prune x
     ; case x' of
         (S y) ->
           do { t <- (seqmap  col y)
              ; return (S t)}
         (MutVar r) -> return(MutVar r)
         (GenVar n) -> return(GenVar n)}
occursIn v t =
  do { t2 <- prune t
     ; case t2 of
         S w ->
           do { s <- (seqmap (occursIn v) w)
              ; return(accRS lib (||) s False)
              }
         MutVar z -> return(sameVarRS lib v z)
         GenVar n    -> return False }
varBind r1 t2 =
          do { b <- occursIn r1 t2
             ; if b
                   then errorRS lib "OccursErr"
          else write r1 (Just t2) }
unify tA tB =
  do { t1 <- prune tA
     ; t2 <- prune tB
     ; case (t1,t2) of
         (MutVar r1,MutVar r2) ->
           if sameVarRS lib r1 r2
              then return ()
              else write r1 (Just t2)
         (MutVar r1,_) -> varBind r1 t2
         (_,MutVar r2) -> varBind r2 t1
         (GenVar n,GenVar m) ->
           if n==m
              then return ()
              else errorRS lib "Gen error"
         (S x,S y) ->
           case matchRS lib x y of
             Nothing -> errorRS lib "ShapeErr"
```

```
                  Just pairs ->
                    mapM_ (uncurry unify) pairs
              (_,_) -> errorRS lib
"ShapeErr"
          }
    match tA tB =
      do { t1 <- prune tA
         ; t2 <- prune tB
         ; case (t1,t2) of
               (MutVar r1,_) ->
                 write r1 (Just t2)
               (GenVar n,GenVar m) ->
                 if n==m
                    then return ()
                    else errorRS lib "Gen error"
               (S x,S y) ->
                  case matchRS lib x y of
                    Nothing -> errorRS lib "ShapeErr"
                    Just pairs ->
                      mapM_ (uncurry match) pairs
               (_,_) -> errorRS lib "ShapeErr"
         }
    equal x y =
      case (x,y) of
        (MutVar r1,MutVar r2) ->
          sameVarRS lib r1 r2
        (GenVar n,GenVar m) -> m==n
        (S x,S y) ->
          case matchRS lib x y of
            Nothing -> False
            Just pairs -> all (uncurry equal) pairs
        (_,_) -> False
    inst sub x =
      do { x' <- prune x
         ; case x' of
              MutVar r -> return(MutVar r)
              GenVar n -> col (sub !! n)
              S x ->
                do { x' <- (seqmap (inst sub) x)
                   ; return (S x')
         }         }
    fromFix (Fix x) = S(mapRS lib fromFix x)
    toFix x =
      do { x' <- prune x
         ; case x of
```

```
             MutVar r -> errorRS lib "No vars"
             GenVar m -> errorRS lib "No generic vars"
             S y -> do { y' <- seqmap toFix y
                       ; return(Fix y')
                       }}
```

B Command language example

```
type Var = String
type Exp = Bool

 data C x
   = If Exp x x        --  if e then s2 else s1
   | While Exp x       --  while e do s
   | Begin x x         --  { s1 ; s2 }
   | Skip              --  {}
   | Assign Var Exp    --  x := e

-- the lowercase constructor convention
ifc e x y = S(If e x y)
while e x = S(While e x)
begin x y = S(Begin x y)
skip = S Skip
assign v e = S(Assign v e)

type Command a = GT C (STRef a)

---- The S structure operators

mapC f (If e x y) = If e (f x) (f y)
mapC f (While e x) = While e (f x)
mapC f ( Begin x y) =  Begin (f x) (f y)
mapC f Skip = Skip
mapC f (Assign v e) = Assign v e

accC acc (If e x y) ans = acc x (acc y ans)
accC acc (While e x) ans = acc x ans
accC acc ( Begin x y) ans = acc x (acc y ans)
accC acc Skip ans = ans
accC acc (Assign v e) ans = ans

seqC (If e x y) =
  do { x' <- x; y' <- y; return (If e x' y')}
seqC (While e x) =
  do { x' <- x; return(While e x')}
```

```
seqC ( Begin x y) =
  do { x' <- x; y' <- y; return( Begin x' y') }
seqC Skip = return Skip
seqC (Assign v e) = return(Assign v e)

matchC (If e w x) (If f y z) =
  if f==e then Just[(w,y),(x,z)]
          else Nothing
matchC (While e w) (While f y) =
  if f==e then Just[(w,y)]
          else Nothing
matchC (Begin w x) (Begin y z) = Just[(w,y),(x,z)]
matchC Skip Skip = Just[]
matchC (Assign v e) (Assign u f) =
  if v==u && e==f
     then Just []
     else Nothing
matchC _ _ = Nothing
```

References

Cardelli, L. (1987) Basic polymorphic typechecking. *Sci. Comput. Program.* **8**(2), 147–172.

Harper, R. and Lillibridge, M. (1994) A type-theoretic approach to higher-order modules with sharing. *Conference Record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 123–137. Portland, OR. ACM Press.

Harper, R. and Pierce, B. C. (2000) Advanced module systems (invited talk): a guide for the perplexed. *ACM SIGPLAN Notices*, **35**(9), 130–130.

Hinze, R. (1999) A generic programming extension for Haskell. *Proceedings of the Third Haskell Workshop, Paris France, Sept. 1999*, University of Utrecht, Technical Report, UU-CS-1999-28.

Hinze, R. (2000a) A new approach to generic functional programming. *Proceedings 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00)*, pp. 119–132. ACM Press.

Hinze, R. (2000b) Memo functions, polytypically! In: Jeuring, J. (editor), *Proceedings 2nd Workshop on Generic Programming, WGP'2000*, pp. 119–132. Ponte de Lima, Portugal.

Jansson, P. and Jeuring, J. (1997) PolyP — a polytypic programming language extension. *Conference Record of POPL '97, 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 470–482. Paris, France. ACM Press.

Jansson, P. and Jeuring, J. (1998) Functional pearl: Polytypic unification. *J. Functional programming*, **8**(5), 527–536.

Jeuring, Johan, & Jansson, P. (1996) Polytypic programming. In: Launchbury, J., Meijer, E. and Sheard, T. (editors), *Tutorial Text 2nd International School on Advanced Functional Programming: Lecture Notes in Computer Science 1129*, pp. 68–114. Springer-Verlag.

Jones, M. (2000) Type classes and functional dependencies. *Proceedings 9th European Symposium on Programming, ESOP 2000: Lecture Notes in Computer Science 1782*. Springer-Verlag.

Jones, M. P. (1996) Using parameterized signatures to express modular structure. *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pp. 68–78. St. Petersburg Beach, Florida.

Jones, M. P. (1999) Typing Haskell in Haskell. *Proceedings Haskell Workshop*, pp. 68–78. (Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht.)

Lämmel, R. and Peyton Jones, S. (2003) Scrap your boilerplate: a practical design pattern for generic programming. *Proc. ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*. ACM Press.

Lämmel, R. and Visser, J. (2000) Type-safe functional strategies. *Draft Proceedings SFP'00, St Andrews*.

Lämmel, R. (2002) Typed generic traversal with term rewriting strategies. *J. Logic & Algebraic Program.* **54**(September). (Also available as arXiv technical report cs.PL/0205018.)

Launchbury, J. and Peyton-Jones, S. (1994) Lazy functional state threads. *PLDI'94: Programming Language Design and Implementation*, pp. 24–35. Orlando, FL. ACM Press.

Leijen, D. and Meijer, E. (2001) *Parsec: Direct style monadic parser combinators for the real world*. Technical report UU-CS-2001-35, Departement of Computer Science, Universiteit Utrecht.

Liang, S. and Hudak, P. (1996) Modular denotational semantics for compiler construction. *Programming Languages and Systems – ESOP'96, Proc. 6th European Symposium on Programming: Lecture Notes in Computer Science 1058*, pp. 219–234. Springer-Verlag.

Liang, S., Hudak, P. and Jones, M. (1995a) Monad transformers and modular interpreters. *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 333–343. ACM.

Liang, S., Hudak, P. and Jones, M. (1995b) Monad transformers and modular interpreters. *Conference Record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 333–343. ACM Press.

Odersky, M. and Läufer, K. (1996) Putting type annotations to work. *Conference Record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 54–67. St. Petersburg Beach, FL. ACM Press.

Sheard, T. (2001) Generic unification via two-level types and parameterized modules. In: Norris, C. and Fenwick, Jr. J. B. (editors), *Proceedings Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP-01)*, pp. 86–97. ACM Press. (Also in *ACM SIGPLAN Notices*, **36**(10).)

Sheard, T. and Peyton Jones, S. (2002) Template meta-programming for Haskell. *Proceedings Workshop on Haskell*, pp. 1–16. ACM Press.

Sheard, T., El-Abidine Benaissa, Z. and Pasalic, E. (1999) DSL implementation using staging and monads. *Domain-Specific Languages, Proceedings of the 2nd Conference on Domain Specific Languages. Austin, Texas, Oct. 3–5, 1999*, 81–94. (Reprinted as Sigplan Notices, Vol 35, No. 1, Jan. 2000.)

Steele, Jr., G. L. (1994) Building interpreters by composing monads. *Conference Record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 472–492. Portland, OR. ACM Press.

Taha, W. and Sheard, T. (1997) Multi-stage programming with explicit annotations. *Partial Evaluation and Semantics-based Program Manipulation*, pp. 203–217. Amsterdam, The Netherlands. ACM.

Winstanley, N. (1997) A type-sensitive preprocessor for Haskell. *Glasgow Workshop on Functinal Programming*, Ullapool, Scotland.